Scalable RDMA performance in PGAS languages

Montse Farreras[†], George Almási[‡], Călin Caşcaval[‡], Toni Cortes[†]

[†] Department of Computer Architecture, Universitat Politecnica de Catalunya
Barcelona Supercomputing Center, Barcelona, Spain
{mfarrera, toni}@ac.upc.es
[‡] IBM T.J. Watson Research Center, Yorktown Heights, NY
{gheorghe, cascaval}@us.ibm.com

Abstract

Partitioned Global Address Space (PGAS) languages provide a unique programming model that can span shared-memory multiprocessor (SMP) architectures, distributed memory machines, or cluster of SMPs. Users can program large scale machines with easy-to-use, shared memory paradigms.

In order to exploit large scale machines efficiently, PGAS language implementations and their runtime system must be designed for scalability and performance. The IBM XLUPC compiler and runtime system provide a scalable design through the use of the Shared Variable Directory (SVD). The SVD stores meta-information needed to access shared data. It is dereferenced, in the worst case, for every shared memory access, thus exposing a potential performance problem.

In this paper we present a cache of remote addresses as an optimization that will reduce the SVD access overhead and allow the exploitation of native (remote) direct memory accesses. It results in a significant performance improvement while maintaining the run-time portability and scalability.

1. Introduction

Parallel programming for both multicores and large scale parallel machines is becoming evermore challenging; adequate programming tools offering both ease of programming and productivity are essential. However, while the productivity of developing applications for these machines is important, the users of massively parallel machines are expecting the same level of performance as obtained by manual tuning of MPI applications.

Partitioned Global Address Space (PGAS) languages, such as UPC, Co-Array Fortran, and Titanium, extend existing languages (C, Fortran and Java, respectively) with constructs to express parallelism and data distributions. These languages provide a simpler, shared memory-like programming model, with control over the data layout. The performance of these languages relies on two main factors: (i) the programmer to tune for locality by specifying appropriate data layouts; and (ii) the compiler and runtime system to efficiently implement the locality directives. The discussion in this paper will focus on the UPC language, however, the optimizations apply to any of the PGAS languages.

The memory model of UPC follows the PGAS model, with each thread having access to a private, a shared local, and a shared global section of memory. Threads have exclusive, low latency, access to the private section of memory. Typically the latency to access the shared local section is lower than the latency to access the shared global section. The UPC memory and threading model can be mapped to either distributed memory machines, shared memory machines or hybrid (clusters of shared memory machines, such as MareNostrum [3]).

In this paper we describe an optimization in the IBM XLUPC runtime system to exploit hardware features, such as RDMA, to improve application performance and maintain a scalable design. The IBM XLUPC runtime system uses a Shared Variable Directory (see Section 2) to provide a scalable infrastructure that has been demonstrated to scale to hundreds of thousands of threads on the BlueGene/L computer [8]. However, in this scalable design, nodes keep only local information. To exploit RDMA, we implemented a cache of remote addresses on two different platforms: the MareNostrum [3] supercomputer, and a cluster of Power5 SMP machines. We measured the performance of this optimization on a set of benchmarks, and we demonstrate a reduction in execution time of up to 40% and 30%, on each of the platforms, respectively.

The effects of using RDMA are the largest for very short messages; these are the kinds of performance improvements that conventional two-sided messaging systems cannot achieve because of design limitations (e.g. MPI point to point relies on message matching on the receiver, which rules out RDMA transfers).

This paper makes the following main contributions:

- describes the Shared Variable Directory (SVD), which is crucial to the scalability of the UPC Run-Time System (RTS).
- presents a runtime optimization that improves its

performance while being scalable and portable.

- introduces a UPC parallel implementation of a subset of the DIS Stressmark Suite and evaluates these benchmarks on our system.

The rest of the paper is organized as follows. Section 2 describes XLUPC RTS. Section 3 describes the runtime optimization implemented to speed up communication. Section 4 outlines the experiments and the results obtained from running the benchmarks on MareNostrum supercomputer and the AIX cluster. The related work is presented in Section 5 and finally conclusions and future work are discussed in Section 6.

2. The IBM XLUPC runtime

The runtime component of the IBM XLUPC compiler has multiple roles: it spawns and collects UPC threads, implements accesses to shared data, performs pointer arithmetic on pointers to shared objects and implements all the UPC intrinsic function calls (such as `upc_phaseof`, `upc_barrier` and `upc_memget`). The XLUPC runtime defines an external API that is used by the UPC compiler when generating code.

The overall architecture of the XLUPC runtime (Figure 1) is similar to that of GASNet [6]. It provides a platform-independent interface that can be implemented on top of a variety of architectures, SMP or distributed.

The XLUPC runtime is designed for a *hybrid* mode of operation on clusters of SMP nodes: UPC threads communicate through shared memory when available, and send messages through one of several available transports where necessary. The Pthreads library is used to spawn multiple UPC threads on systems with SMP nodes. Implemented messaging methods include TCP/IP sockets, LAPI [23], Myrinet/GM transport [19] and the BlueGene/L messaging framework [1].

In this paper we focus on optimizing the hybrid and distributed-memory implementations of the runtime.

2.1. The Shared Variable Directory

Shared objects (i.e. data structures accessible from all UPC threads) form the basis of the UPC language. The XLUPC runtime recognizes several kinds of shared objects: *shared scalars* (including structures/unions/enumerations), *shared arrays* (including multi-blocked array [7]), *shared pointers* (with either shared or private targets) and *shared locks*.

All UPC shared objects have an *affinity* property. A shared object is affine to a particular UPC thread if it is local to that thread's memory. Shared arrays are distributed in a block-cyclic fashion among the threads, so different pieces of the array have affinity to different threads.

Access to shared objects presents a scalability problem that all UPC implementations share, namely that the addresses of locally allocated portions of shared objects are needed by other nodes in order to access shared data. There are multiple solutions to address this problem:

- Ensure that shared objects have the same addresses in all nodes. Unfortunately this approach does not work too well with dynamic objects: it tends to fragment the address space and it is cumbersome to implement, sometimes requiring changes to the system memory allocator.
- A distributed table of size $O(nodes \times objects)$ can be set up to track the addresses of every shared object on every node. For a large number of nodes or threads, this can be prohibitively expensive and thus, directly impacting scalability. It also requires extensive communication when the virtual to physical mapping changes on a particular node.
- The third solution involves a distributed symbol table. Shared objects are known by their handles (unique identifiers for every shared object). Translation from shared object handles to memory locations only happens on the home node of the shared object.

In the XLUPC runtime we opted for the last approach. Shared objects are organized into a distributed symbol table called the Shared Variable Directory (SVD). The SVD manages the life cycle of shared objects (allocation, freeing, use). On a system with n UPC threads the SVD consists of $n + 1$ partitions. Partition k, $0 \leq k < n$ holds a list of those variables *affine* to thread k. The last partition (called the *ALL* partition) is reserved for shared variables allocated statically or through collective operations.

Each partition in the SVD is a list of shared objects. Shared objects that are locally allocated have an associated control structure containing the memory addresses in question (Figure 2).

Shared objects are referred to by their *SVD handles*, opaque objects that internally index the SVD. An SVD handle contains the **partition** number in the directory, and the **index** of the object in the partition.

Multiple replicas of the SVD exist in a running XLUPC system. The SVD often changes at runtime because of UPC routines for dynamic data allocation, such as `upc_global_alloc`, `upc_all_alloc`, and `upc_local_alloc`. The SVD has to be kept internally consistent. Partitioning greatly aids this process, because it allows the SVD to be kept consistent with a minimal effort and without any bottlenecks:

1) UPC threads can allocate and de-allocate shared variables independently of each other. Each thread updates its own partition, and sends notifications to other threads;
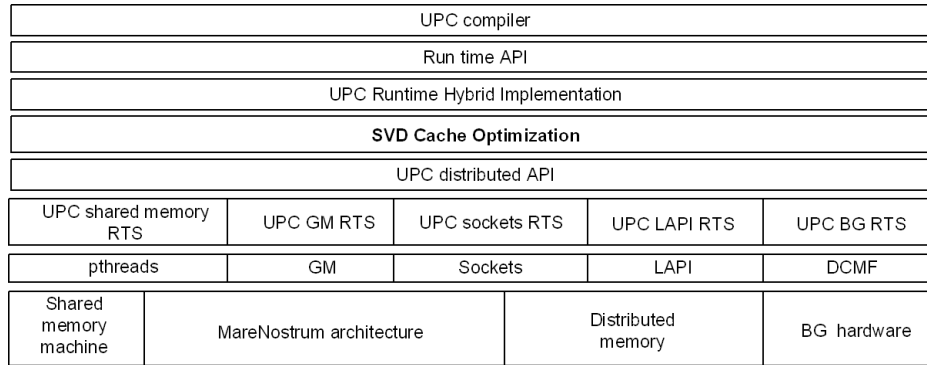
2

UPC compiler				
Run time API				
UPC Runtime Hybrid Implementation				
SVD Cache Optimization				
UPC distributed API				
UPC shared memory RTS	UPC GM RTS	UPC sockets RTS	UPC LAPI RTS	UPC BG RTS
pthreads	GM	Sockets	LAPI	DCMF
Shared memory machine	MareNostrum architecture		Distributed memory	BG hardware

Figure 1: Software organization of XLUPC runtime

2) Because the SVD is partitioned, and each partition has a single writer, memory allocation events do not require locks. The *ALL* partition is only updated by collective memory allocation operations that are already synchronized.

2.2. The performance compromise of the SVD design

We consider the SVD essential to the scalability of the XLUPC runtime. Unfortunately there is a price to be paid for translating SVD handles to memory addresses only at the target node.

Modern communication transports (like Myrinet [19] and LAPI [15]) have one-sided RDMA communication primitives that require no CPU involvement on the remote end. However, these primitives typically require the *physical* address of the shared object at the target to be known by the initiator of the communication. Figure 3 contrasts the implementation of remote GET when the address of the object is known (b), and when it is not (a).

RDMA GET and PUT functions cannot be used in a naive SVD implementation, since SVD lookups require CPU involvement. This both lengthens communication latency and burdens the target CPU with work, contributing to the scalability bottleneck.

In the next section we outline an optimization that allows the use of RDMA operations while preserving the SVD design, by caching the remote addresses of shared objects on an as-needed basis.

3. Remote Address Cache

The goal of the remote address cache (hereafter `address cache`) is to enable small message transfers via RDMA, and thus reduce latency and improve the performance of the XLUPC runtime system. It does

this by caching remote addresses as needed by computation. Unlike a full table of remote addresses, the cache's memory requirements can be controlled and offset against performance.

The `address cache` is implemented as a hash table. Each entry in the cache correlates a universal SVD handle and a node identifier ID with the physical base address for the shared variable identified by the SVD handle on the remote node ID. During a GET or PUT operation the initiating node consults the `address cache` for the base address on the target node. A cache hit guarantees that the final remote address (base address + offset) can be calculated on the initiator node, allowing the message transfer to be executed as an RDMA operation.

Conversely, if the address lookup in the cache fails the operation cannot be executed as an RDMA operation. However, the slower operation can be harnessed to retrieve the remote base address for the next operation to the same node. We have modified the default (non-RDMA) one-sided messaging protocol to retrieve the base address of the remote shared object during the transfer by piggybacking it either on the data stream or on the ACK message.

To ensure the correctness of RDMA transfers, the remote node has to guarantee that the remote memory address has not changed between accesses. To ensure that the *physical* address (typically required by RDMA operations) of a shared array is fixed, we need to *pin* the array in memory [28], [24]. To this end we augmented the address cache with a table of registered (pinned) memory locations. The *pinned address table* is tagged by **local** virtual addresses and contains physical addresses in the format needed by RDMA operations.

Figure 4 portrays a typical runtime snapshot. In the figure the address cache of node A caches two remote addresses on node B. Both these entries exist in the pinned address table of node B. Node C also caches
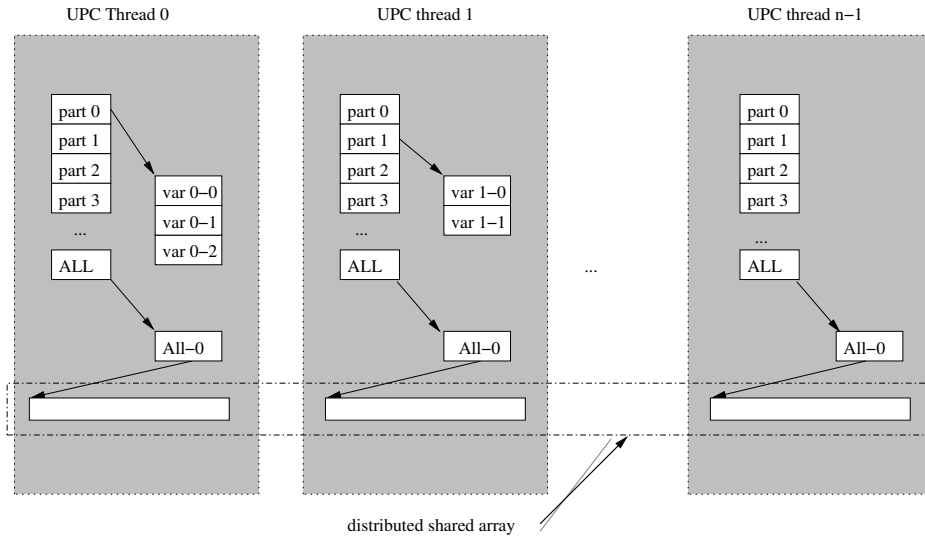
3

Figure 2: Shared Variable Directory in a distributed memory machine. Gray boxes represent the local memory of each of n UPC threads, each with a copy of the SVD. The SVD has $n + 1$ partitions; each partition has a list of control blocks, one each for shared objects known locally. Addresses are only held for the local or ALL partitions. Distributed shared array "All-0" has a different local address on every node.
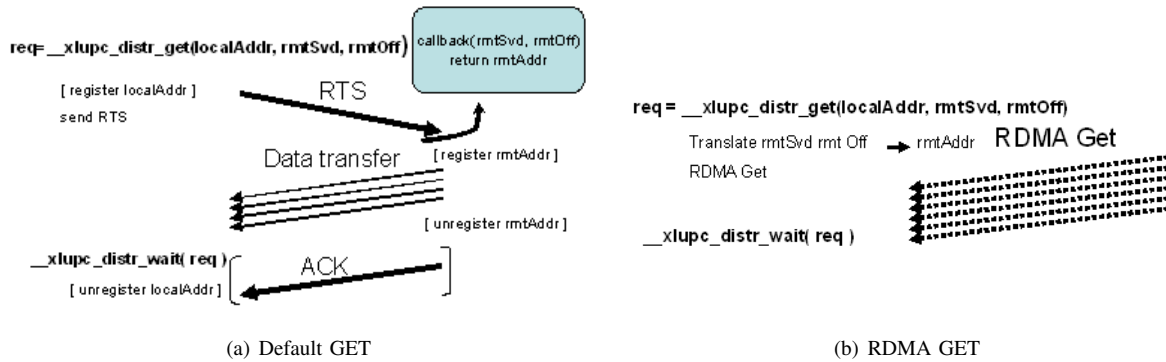

(a) Default GET

(b) RDMA GET

Figure 3: Protocols for GET: (a) when the remote address is unknown a Request To Send (RTS) message needs to be sent before every data transfer. Acknowledge message may be required depending on the implementation (i.e. to signal completion or to carry the remote address if data transfer is performed thought one-sided communication). And (b) with RDMA operation when the remote address is known.

one of these entries, as well as another entry on node A (shaded appropriately to show the correspondence).

3.1. Implementation details

In our implementation, before an address can be tagged in another node's `address cache` it needs to be pinned locally. Nevertheless, the pinning strategy is not the goal of this paper and for the sake of simplicity a greedy "pin everything" approach is presented here. We resolve that: (i) the entire memory allocated for a shared object is pinned at once on a particular node. For example, if any element of a shared array A is accessed on node n, *all* of the memory related to array A is pinned on that node, making it all available for RDMA

operations. And also, (ii) once a shared object is pinned it remains pinned until it is freed. The address cache is eagerly invalidated when a shared object is deallocated.

These decisions simplify the implementation of the address cache, because the cache tags can simply be the SVD handles and consistency of the remote `address cache` is not an issue.

However, these assumptions limit the implementation in two important respects. First, it does not deal with network transports that limit the amount of contiguous memory pinned by a single call, and also limits on the total amount of memory pinned are ignored. We have successfully implemented a more elaborated technique to deal with both these issues obtaining similar results.
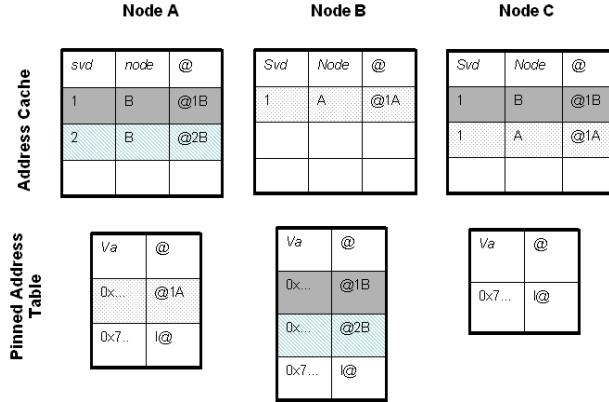
4

Figure 4: Example snapshot of address cache on three nodes.

A detailed explanation can be found in [10].

3.2. Considerations for a LAPI based implementation

The LAPI implementation of the XLUPC low level messaging API uses Active Messages (`LAPI_Amsend`) (as shown in Figure 5 (a)). Implementation of the remote address cache required only trivial changes in the messaging library to enable the process of populating the `address cache`. Cache hits result in messages that bypass the standard messaging system completely and use RDMA directly.

LAPI limits the amount of memory that can be assigned to a single registered memory handle (a configuration parameter, 32 MBytes on our machines).

3.3. Considerations for the Myrinet/GM implementation

GM is a message-based communication system for Myrinet [19]. The XLUPC runtime Myrinet port was implemented on top of GM instead of MX, mainly because GM is currently the driver installed on our testing platform (the MareNostrum supercomputer [3]). Moreover, GM provides one-sided support and primitives to directly expose the RDMA capabilities of the hardware.

However, three considerations need to be taken into account: first, Myrinet requires memory registration for any data transfer, and it is handled by the programmer in GM. Second, the largest message GM can send or receive is limited by the amount of DMAable memory the GM driver is allowed to allocate by the operating system. (1 GByte on our test machines). Finally, memory registration is an expensive operation; memory de-registration even more so.

Our original XLUPC Myrinet port implements multiple transfer protocols depending on message size [20]. Short messages are copied to avoid memory registration costs. Long messages are transferred with an MPI-like rendezvous protocol with memory registration/de-registration embedded into the phases of the protocol. As an optimization a cache of registered memory regions was implemented [26] with lazy memory de-registration.

Again, the changes required in the messaging library to enable the remote address cache were trivial (Figure 5 (b)) and very similar to the ones in LAPI.

4. Evaluation

The scalability of the XLUPC runtime with the Shared Variable Directory has been demonstrated in our previous work [8]. In this paper we aim to demonstrate the effect of the `address cache` on messaging performance. Our experiments were performed on comparatively smaller machines: 512 nodes of the MareNostrum supercomputer and a 28-node Power5 cluster running AIX.

We defined a confidence coefficient of 95% and ran each experiment multiple times to reduce the standard error. We assumed experiments to be independent, therefore the formulas associated with a normal distribution apply [14].

4.1. Evaluation environment: MareNostrum

MareNostrum [3] is a cluster of 2560 JS21 blades, each with two dual core IBM PPC 970-MP processors sharing 8 GBytes of main memory. The processors are equipped with a 64 KByte instruction/32 KByte data L1 cache and a 1024 KBytes of L2 cache and run the SLES9 (Linux) operating system.
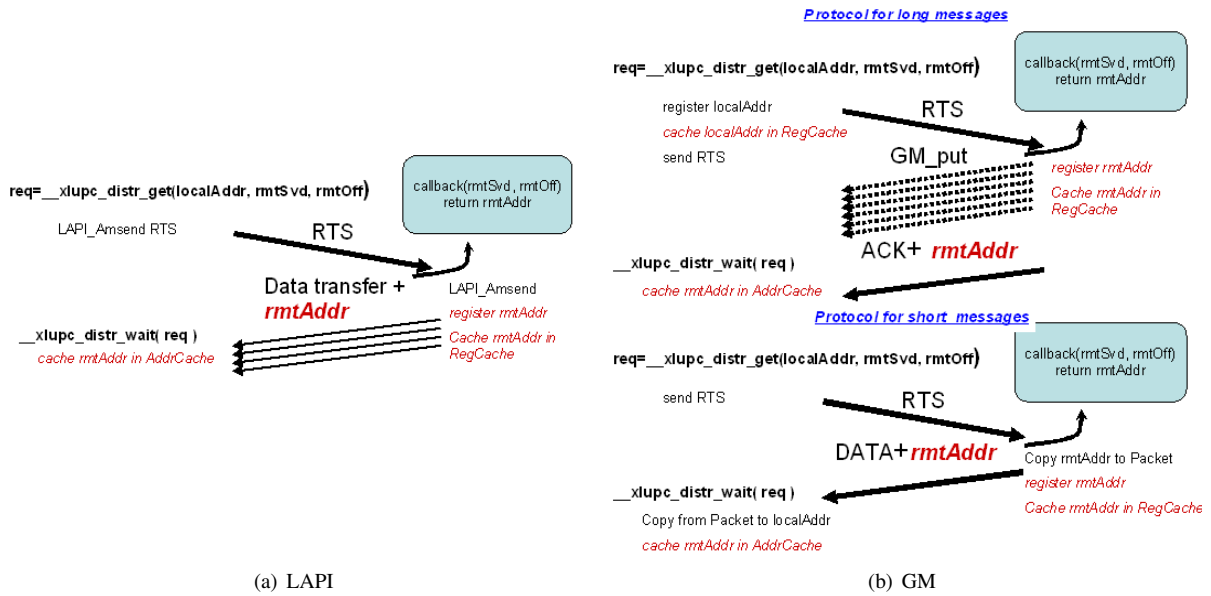
5

(a) LAPI   (b) GM

Figure 5: Enabling `address cache` population in the standard XLUPC implementation of GET (changes shown in *italics*), in (a)LAPI and (b)GM. The initiating process sends an Active Message that triggers the execution of a header handler on the passive target. The header handler performs address translation *and memory registration*. The reply sent back to initiator process contains requested user's data *plus the address*. GM requires different protocols for different message sizes.

The MareNostrum's interconnection network is Myrinet with a 3-level crossbar, resulting in 3 different route lengths (1 hop, when two nodes are connected to the same crossbar aka. *linecard*, and 3 hops or 5 hops depending on the number of intervening linecards). As mentioned before, we implemented the XLUPC runtime transport on top of the Myrinet/GM messaging library.

4.2. Evaluation environment: Power5 cluster

The second machine we performed our experiments on is a 28-node cluster of Power5 servers. Each node is with 16 GBytes of RAM and 8 2-way SMT Power5 cores running at 1.9 GHz. The nodes are connected with an IBM High-Performance Switch (HPS). The operating system is AIX 5.3; we used the LAPI library provided as part of the Parallel Operating Environment on the system as the basis for the XLUPC transport implementation.

4.3. Microbenchmarks

Our first set of experiments sought to quantify the maximum benefit obtainable by the address cache. We wrote and executed microbenchmarks to compare GET roundtrip latencies and PUT overheads of the XLUPC runtime with and without cache operation.

Figure 6 shows the relative gains (i.e. execution time reduction) caused by deploying the `address cache`

for PUT and GET on both LAPI and Myrinet/GM transports. We see three distinct modes on all platforms.

- For very small messages (up to 1 KByte in size) the gains in GET roundtrip latency (left panel of Figure 6) are in 30% and 16% range respectively for GM and LAPI. This is the optimization that we had been targeting.
- For medium message size range messages (1 KByte to 16 KByte) there are even larger gains (around 40%). This is likely due to the rising cost of memory copies in the non-cached case. The gain is more visible on LAPI, fadding out at 2 MByte, than on Myrinet because the rated bandwidth of the HPS switch is 8x that of Myrinet. Roundtrip latencies of both networks are in the 4-8 microsecond range. Thus, a fixed amount of overhead reduction (of the order of the roundtrip latency) affects the HPS switch for much higher message sizes than Myrinet.
- As expected, differences between cached and un-cached behavior diminish as message size increases and communication becomes bandwidth dominated instead of overhead dominated.

With PUT messages (right panel of Figure 6), in GM we do not see any benefit of using the `address cache` for small message transfers, up to 2 KBytes. PUT execution time is not significantly affected by address translation overheads, and the load on the remote
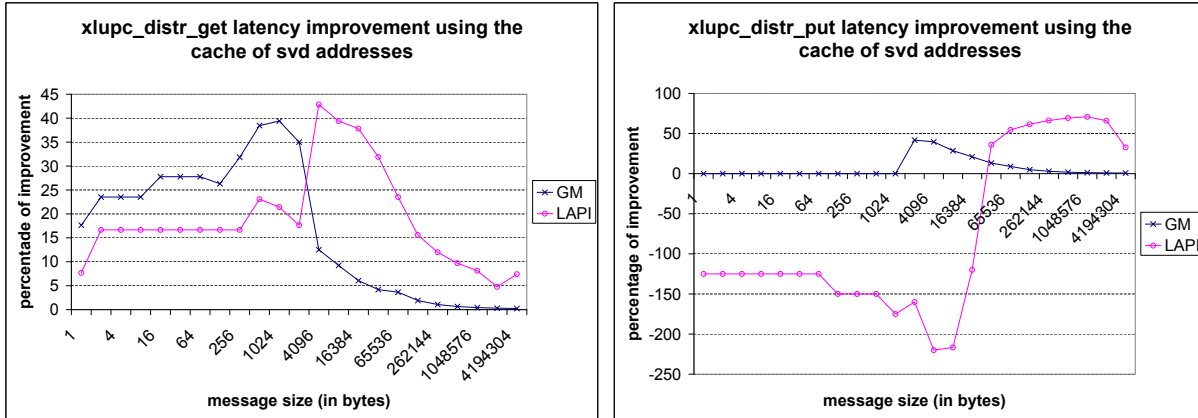
6

Figure 6: Latency improvement by using the `address cache` in both platforms: LAPI(o) and GM (x) considering different message sizes. The Y axis represents the performance benefit in terms of execution time reduction by using the `address cache`, expressed in percentage ($\frac{100(Z-W)}{Z}$, with Z being the average regular latency, and W being the average latency by making use of the `address cache`. X axis shows the message size.
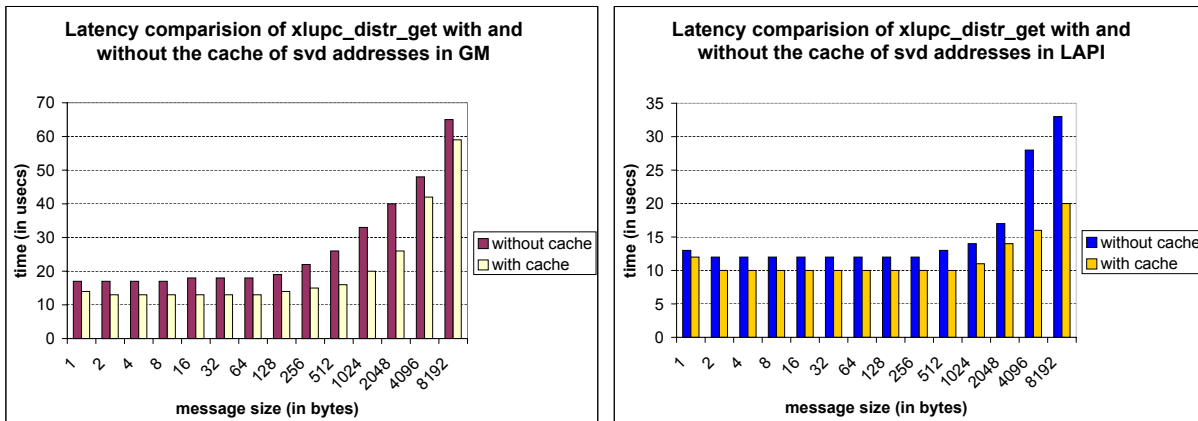

Figure 7: GET latency with and without the `address cache` in both platforms: GM and LAPI, considering short message sizes. The Y axis represents the latency. The X axis is the message size.

CPU is not measured on this microbenchmark.

The situation is different for LAPI, where we see a net decrease in performance of up to 200% by using the `address cache`. The cause of this performance decrease is the IBM switching hardware, which offers excellent throughput in RDMA mode, at the cost of higher latency. In a GET operation the higher latency is offset by the fact that the remote node's CPU time is not part of the roundtrip time; in the case of PUT the remote CPU's operation is overlapped with the next send. Following these results, we disabled the `address cache` for the PUT operations in LAPI.

Figure 7 shows another view of the GET roundtrip latency data: the absolute latencies in microseconds for small message GET operations with and without address caching on both our test platforms.

4.4. DIS Stressmark Suite

We have implemented a subset of the DIS Stressmark Suite [2] and ran it with and without address caching. DIS benchmarks have been chosen over e.g. the NAS benchmarks because they recreate the access patterns of *data-intensive* real applications, whereas most NAS benchmarks are compute- rather than data-intensive. Four of the benchmarks have been implemented and evaluated (Figure 9):

- The **Pointer Stressmark** is repeatedly following pointers (hops) to randomized locations in memory until a condition becomes true. The entire process is performed multiple times. Each UPC thread runs the test separately with different starting and ending positions on the same shared array.
- The **Update Stressmark** is a pointer-hopping benchmark similar to the *Pointer Stressmark*. The major difference is that in this code more than one

7

remote memory location is read - and one remote location is updated - in each hop. All this is done by UPC thread 0, while the other threads idle in a barrier. This benchmark is designed to measure the overhead of remote accesses to multiple threads.

- The **Neighborhood Stressmark** is a stencil code prototype. It deals with data that is organized in multiple dimensions. It requires memory accesses to pairs of pixels with specific spatial relationships. Computation is performed in parallel based on the locality of the shared array. The two-dimensional pixel matrix is block-distributed in a row major fashion. Accesses are local or remote depending on stencil distances and pixel positions.
- The **Field Stressmark** emphasizes regular access to large quantities of data. It searches an array of random words for token strings, that delimit the sample sets, from which simple statistics are collected. The delimiters themselves are updated in memory. When all instances of a token are found and statistics computed, the process is repeated with a different token.

  The string array is blocked in memory (i.e. with a block size of $\lceil \frac{N}{THREADS} \rceil$, with N being the array size). Because the array is updated in every run, the outermost loop (which iterates over multiple tokens) cannot be parallelized. Parallelization is done instead in the inner loop, where each UPC thread searches the local portion of the data string for tokens. Because a token may span the boundary of two segments affine to different threads, the threads must overlap their search spaces by at least the width of a token to guarantee that all tokens are found.

  Since token lengths are typically much smaller than the string array's blocking factor, most accesses in the algorithm are local.

4.5. Cache size considerations

Cache size is an important metric that may affect overall application performance. The space needed by the `address cache` depends on both the number of shared variables declared by the UPC application and the communication pattern in the running application. Most UPC applications (with a few notable exceptions) declare a relatively small number of shared variables and have static and well defined communication patterns that result in insignificantly small caches even on large machines.

Our selected subset of benchmarks covers both types of applications. Field and Neighborhood follow the well defined communication patterns of typical UPC applications. As it is shown for Neighborhood in figure 8

(b), it results in insignificantly small caches, only a few cache entries are used and the hit ratio keeps constant as we scale. Whereas Pointer and Update belong to the group of rare UPC applications that unpredictably access remote memory locations along the whole shared memory space, which results in address caches that grow with the number of nodes. Figure 8 (a) shows for Pointer hit ratio degradation as we scale, with a prompt starting point as cache size is reduced. For this kind of applications we have a compromise between memory usage and speedup.

The Address Cache is currently implemented as a dynamic hash table. Its size is allowed to increase on demand to a fixed limit of 100 entries. The next section shows a considerably performance improvement, even for applications following an unpredictable communication pattern.

Concerning the *Pinned Address Table*, its appropriate size depends exclusively on the number of shared variables. It is our experience that in most UPC programs (i.e. the NAS benchmarks, the HPC Challenge benchmarks, a suite of NSA benchmarks and others) the number of shared variables is relatively small. Shared variables enhance productivity but decrease performance; most good UPC programs therefore limit their number. Our experiments show that a table of 10 entries is more than enough for well defined UPC applications.

4.6. Stressmark evaluation on MareNostrum

We evaluated the 4 Stressmarks on a 512 blade subset of the MareNostrum computer, with 4 UPC threads running on each blade. A representative subset of our experiments is shown in Figure 9.

Since UPC threads running in the same blade share memory, remote communication between these threads does not involve the network hardware. This can affect measured performance improvements, since no benefit from the `address cache` can be expected when the network hardware is not in use.

The *Pointer Stressmark* shows good performance, between 30% and 60% improvement depending on the total number of remote accesses in the benchmark. The performance of the Pointer Stressmark is gated by network latencies and overheads; any reduction in these results in performance gains. In hybrid execution mode the network device is shared by all UPC threads running on a blade; the improvement caused by smaller CPU overheads was augmented by smaller network device overheads; with four threads competing for the same network device any improvement in network device access time is magnified fourfold. Since in cached mode the DMA is doing the work of data transfer, the four threads spend much less time queueing up for access to
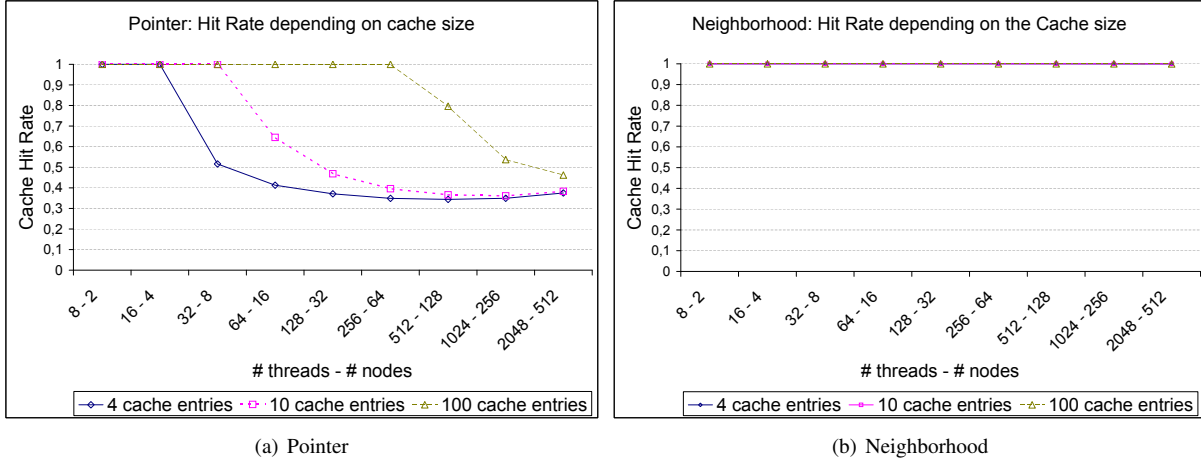
8

(a) Pointer

(b) Neighborhood

Figure 8: Address Cache Size Evaluation using DIS Stressmark Suite. The vertical axes show the percentage of hits on the Address Cache. The horizontal axes of the graphs show the threads and nodes used. Each line represents a different cache configuration in terms of cache size. Results are shown considering a random thread.
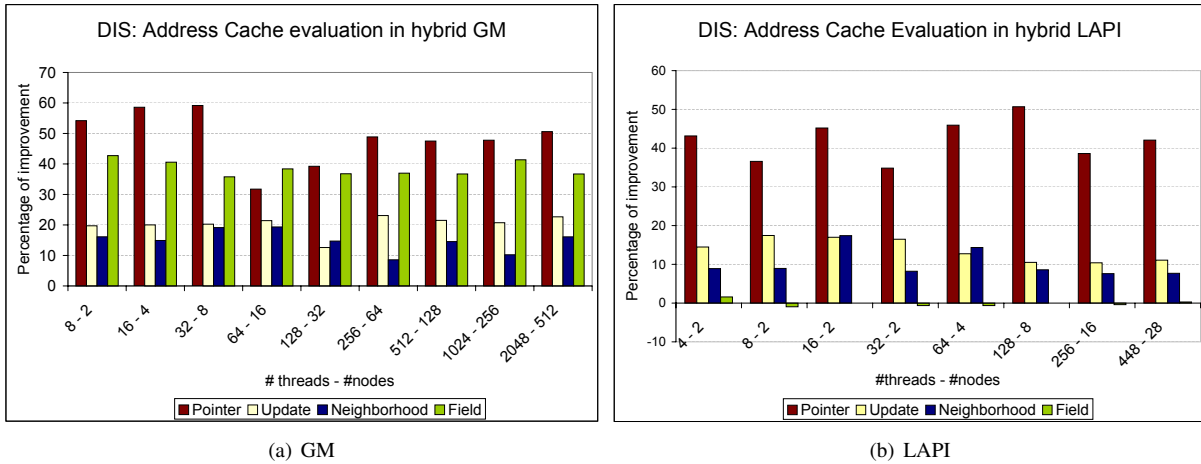

(a) GM

(b) LAPI

Figure 9: Address Cache Evaluation on GM (a) and LAPI (b) using DIS Stressmark Suite The vertical axes show performance improvement when using the remote address cache: $\frac{100(Z-W)}{Z}$, where `Z` and `W` are the regular and address-cache-enabled runtimes respectively. The horizontal axes of the graphs show the threads and nodes used.

the network device. This effect could not be appreciated in the microbenchmark because it ran on 1 active thread in each node.

The *Update Stressmark* shows a 11% to 22% performance improvement when the `address cache` is enabled, which corresponds to the performance measured by the PUT and GET microbenchmarks. We do not see performance improvement caused by two threads per node, because only thread 0 initiates communication.

The *Neighborhood Stressmark* shows 10% to 20% improvement. The stencil used in this experiment (with a stencil distance of 10) causes about $\frac{3}{16}$ of memory accesses to be potentially remote, depending on the number of nodes, therefore the improvement is mostly

along the lines we expected based on the microbenchmark.

The *Field Stressmark* shows a 35% to 40% performance improvement. We analyzed the behavior of this benchmark using the Paraver performance analysis toolkit [3]. The trace showed that the remote GET and PUT access times at the "overhangs" were abnormally large when `address cache` was not in use. As a reminder, each thread in the benchmark scans the local portion of a distributed array. The scan extends into "overhangs" that belong to the two neighboring threads. With normal XLUPC runtime operation the remote node's CPU is part of every remote access; but the Myrinet/GM transport does not overlap communication and computation. While a CPU is busy with the local

9

portion of its array the network does not make progress, and other CPUs requesting data are forced into long waits.

By contrast, when the address cache is in use, RDMA operations are used for remote accesses. These require no cooperation from the remote node's CPU; therefore remote access wait times decrease significantly, and performance improves.

4.7. Stressmark evaluation on Power5 cluster

We have also run the four Stressmarks on the P5 cluster. We chose several thread/node configurations. Figure 9 (b) shows the results with up to 448 UPC threads.

The *Pointer Stressmark*, *Update Stressmark* and *Neighborhood Stressmark*, show results comparable to the measurements on MareNostrum.

The behavior of the *Field Stressmark* on the LAPI transport is markedly different from the GM measurement. LAPI allows overlap of computation and communication, therefore wait times for PUT and GET operations on remote arrays are not excessive even without `address cache` operation. Since the ratio of remote and local operations is relatively low in this benchmark, the effects of the `address cache` are not measurable.

5. Related Work

Previously existing UPC implementations, such as the Berkeley UPC compiler [5] or MuPC[30] from Michigan Technological University [18], map UPC threads to O/S processes, and each thread maps the entire memory space to the same virtual address, forcing virtual addresses to be identical on all threads. This solution results in memory fragmentation (especially when individual threads allocate memory). The XLUPC runtime system prevents this problem by means of the SVD, which allows virtual addresses to be different on each node and allowing UPC threads to be mapped to Pthreads that share memory directly.

Another problem affected by shared variable directories and remote address caching is the handling of memory registration costs on pinning-based networks like Myrinet [19], VIA or Infiniband [21]. MPI implementations like OpenMPI [22] and MVAPICH [27] as well as one-sided messaging systems like ARMCI [20] follow a differential approach based on message size, switching between preallocated registered memory buffers (Bounce Buffers) for short messages and dynamic memory registration and de-registration as part of each data transfer (Rendezvous) for large ones. The crossover point between the protocols is dependent on the underlying network hardware and software, requiring tuning for each machine.

Since on Myrinet/GM the de-registration cost is the most expensive, most existing Myrinet/GM communication layers, including Myricom's own MPICH-GM [19], use Rendezvous and omit the de-registration step.

Another solution to handling memory registration costs is the Pin-Down cacheused in PM [25] and Sockets-GM [19]. The idea of a cache has also been used in UPC for remote data [29].

Berkeley UPC's strategy for on-demand registration of shared memory regions, called Firehose [4], distributes the largest amount of memory that can be registered among all nodes. Every node keeps track both of remote regions in other nodes it is using and its own areas used by other nodes.

The Shared Variable Directory concept and remote addresses caching could potentially be applied to every shared-memory programming model that wants to run on a distributed memory system (e.g. OpenMP on a Software Distributed Shared Memory system). Several combinations of OpenMP runtime plus SDSM systems have been implemented [12]. The most significant ones are the OpenMP translator developed by Hu et al. [13] on top of Treadmarks [17], OpenMP on the SCASH system [11], and ParADE [16]. There is also NanosDSM [9] which uses sequential-semantic memory consistency.

6. Conclusions and future work

In this paper, we have shown how the IBM XLUPC compiler and runtime system provide a scalable design through the use of the Shared Variable Directory (SVD). We have addressed the potential performance problem encountered by short remote memory accesses that need to be dereferenced in the SVD. We have presented a mechanism to cache remote addresses that reduces the SVD access overhead and allows the exploitation of RDMA operations in the network hardware for very short messages, improving latency and scalability.

We have evaluated our proposed optimization on two different platforms: the MareNostrum supercomputer and a Power5 cluster of SMPs, using microbenchmarks and four benchmarks of the DIS Stressmark Suite. Our results demonstrate an average reduction in execution time of up to 40% and 30% respectively on the two architectures. The overhead of unsuccessful attempts to cache remote addresses is relatively small, typically 1.5% and never worse than 2%.

In the future, we plan to extend the range of our scalability experiments to confirm that the performance

benefits we measured on relatively small machine configurations continue into the range of tens of thousands of processors and measure the benefits of the address cache on applications as opposed to benchmarks.

Acknowledgments

References

[1] G. Almasi et al. Design and implementation of message-passing service for the BlueGene/L supercomputer. *IBM Journal of Research and Development*, 49(2/3):393–406, 2005.

[2] Atlantic Aerospace Division, Titan Systems Corporation. DIS Stressmark Suite: Specifications for the Stressmarks of the DIS Benchmark Project. Version 1.0. *MDA972-97-C-0025*, 2000.

[3] Barcelona Supercomputing Center. BSC. http://www.bsc.org.es.

[4] C. Bell and D. Bonachea. A new DMA registration strategy for pinning-based high performance networks. In *17th International Parallel and Distributed Processing Symposium (IPDPS-2003)*, pages 198–198, Los Alamitos, CA, Apr. 22–26 2003. IEEE Computer Society.

[5] Berkeley. UPC Project Home Page, 2005. http://upc.lbl.gov/.

[6] D. Bonachea. Gasnet specification, v1.1. Technical Report CSD-02-1207, U.C. Berkeley, November 2002.

[7] Christopher Barton, Calin Cascaval, George Almasi, Rahul Garg, Jose Nelson Amaral and Montse Farreras. Multidimensional Blocking Factors in UPC. *LCPC2007: International Workshop on Languages and Compilers for Parallel Computing*, 2007.

[8] Christopher Barton, Calin Cascaval, George Almasi, Yili Zheng, Montse Farreras and Jose Nelson Amaral. Shared memory programming for large scale machines. *PLDI 2006: ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006.

[9] J. J. Costa, T. Cortes, X. Martorell, E. Ayguade, and J. Labarta. Running openmp applications efficiently on an everything-shared sdsm. *J. Parallel Distrib. Comput.*, 66(5):647–658, 2006.

[10] M. Farreras. *Optimizing Parallel Models for Massively Parallel Computers*. PhD thesis, Universitat Politcnica de Catalunya (UPC), 2008.

[11] M. Hess, G. Jost, M. Müller, and R. Rühle. Experiences using OpenMP based on compiler directed software DSM on a PC cluster. In *WOMPAT2002*.

[12] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM system based on a new cache coherence protocol. In *Proc. of the High-Performance Computing and Networking Europe 1999 (HPCN'99)*, pages 463–472, 1999.

[13] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. OpenMP for networks of SMPs. *Journal of Parallel and Distributed Computing*, 60(12):1512–1530, 2000.

[14] Hunter W.G., Hunter J.S., Hunter W.G. *Statistics for Experimenters: Design, Innovation, and Discovery*. Box,G. E, Wiley, 2nd Edition, 2005.

[15] IBM. RSCT LAPI Programming Guide, 1990. http://publib.boulder.ibm.com/epubs/pdf/bl5lpg04.pdf.

[16] Y.-S. Kee, J.-S. Kim, and S. Ha. Parade: An openmp programming environment for smp cluster systems. In *SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 6, Washington, DC, USA, 2003. IEEE Computer Society.

[17] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, 1994.

[18] MTU. Michigan technological university, July 2003. http://www.mtu.edu/.

[19] Myrinet Software and Documentation Home Page. Myricom. Myricom: GM, MX, MPICH-GM, MPICH-MX and Sockets-GM. http://www.myri.com/.

[20] J. Nieplocha, V. Tipparaju, and D. Panda. Protocols and strategies for optimizing performance of remote memory operations on clusters. 2002.

[21] D. of Computer Science and E. T. O. S. University. High performance mpi on iba mpi over infiniband project. http://nowlab.cis.ohio-state.edu/projects/mpi-iba/.

[22] OpenMPI. Open mpi: Open source high performance computing. http://www.open-mpi.org/.

[23] G. Shah, J. Nieplocha, J. Mirza, C. Kim, R. Harrison, R. K. Govindaraju, K. Gildea, P. DiNicola, and C. Bender. Performance and experience with lapi - a new high-performance communication library for the ibm rs/6000 sp. In *Proceedings of IPPS '98*.

[24] S. Sur, H.-W. Jin, L. Chai, and D. K. Panda. Rdma read based rendezvous protocol for mpi over infiniband: design alternatives and benefits. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 32–39, New York, NY, USA, 2006. ACM.

[25] H. Tezuka, A. Hori, and Y. Ishikawa. Pm: a highperformance communication library for multi-user parallel environments. In *Usenix'97, 1996.*, 1996.

[26] H. Tezuka, F. O'Carroll, A. Hori, and Y. Ishikawa. Pin-down cache: A virtual memory management technique for zero-copy communication. In *IPPS/SPDP*, pages 308–314, 1998.

[27] The Ohio State University. Mvapich: Mpi over infiniband and iwarp. http://mvapich.cse.ohio-state.edu/overview/mvapich/.

[28] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda. Host-assisted zero-copy remote memory access communication on infiniband. In *IPDPS*. IEEE Computer Society, 2004.

[29] J. S. Wei Chen, Jason Duell. A software caching system for upc. Technical Report CS265 Project, Department of Computer Science, U.C. Berckley, 2003.

[30] Z. Zhang, J. Savant, and S. Seidel. A upc runtime system based on mpi and posix threads. In *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*, pages 195–202, Washington, DC, USA, 2006. IEEE Computer Society.